Amendments to the Specification:

Please amend the paragraph starting at page 48, line 20 as follows:

The procedural model uses a pseudo-code modeled after Concurrent Pascal. IEEE Std 802.3 1998 Clause 4.2.2 provides an overview of this pseudo-code. The code set forth at the end of the specification [[below]] models three independent concurrent processes (Deference, Transmitter, Receiver), which interact through shared variables. The Deference process is driven by the detection of transmissions on the channel, and times the boundaries for Signal Slots and Priority Slots. The shared variable current Priority signals the Transmitter process when a transmission slot exists.

Delete the paragraphs starting from page 48, line 30, through page 53, line 23.

Insert the following new paragraphs after the paragraph ending on page 210, line 7,:

(Deference:
Loop, looking for carrier sense, and when found determine whether the transmission was a collision or valid frame.
If it was a collision, process the signal slots and run the collision resolution algorithm.
In any case, then process the priority slots, looking for carrier.

```
Note that the "current" priority level is sticky from the slot
the last collision occurred in.
Note that the Backoff Level (BL) and Maximum Backoff Level (MBL)
counters  are saturating at 0 and 15.}
Const
     nPriorities = 8;   {Number of priority levels}
     nSignals = 3; {Number of signal slots}
     nLevels = 16; {Number of Backoff Levels}

process Deference;
begin
     currentPriority := 0; {Priority of the slot we are in}
     cycle {deference loop}
          sawFrame := false;
          sawCollision := false;
          while not carrierSense() do nothing; {watch for
carrier to appear}
          deferring := true;
          startTime := time();
          stopTime := startTime;
          while carrierSense() do
               stopTime := time();
          if ((stopTime - startTime > CD_MIN) and
          (stopTime - startTime < CD_THRESHOLD)) or
collisionSense()
               then sawCollision := true
          else sawFrame := true;
          (After a collision, process the three signal slots}
          if sawCollision then
          begin
               {wait until the end of the IFG, timing from start
of fragment
```

```
                reduces skew, since start-of-carrier uncertainty
is less than
                end-of-carrier uncertainty }
               while (time() - startTime < CS_IFG + CD_FRAG) do
nothing();
               computeSignals();
               for (i := 0; i < nSignals; i++)
               begin
                  startTime := time();
                  signal[i] := 0;
                  if signalSlot = i then sendSignal();
                  while (time()-startTime < SIG_SLOT) do
                    if carrierSense() then signal[i] := 1;
               end;
               processSignals();
               end;
               if (not sawCollision) then
               begin
                 {wait until the end of the IFG}
                 while (time() - stopTime < CS_IFG) do
nothing();
                 {If last transmission was successful, drop
Backoff Levels}
                  BL[currentPriority]    :=    saturate(0,nLevels-
1,BL[currentPriority]-1);
                  MBL[currentPriority]    :=    saturate(0,nlevels-
1,MBL[currentPriority]-1);
               end;
               {avoid    timing    hazard    with    transmitter,
currentPriority must be setup
                before deferring is cleared}
                currentPriority := nPriorities-1;
```

```
          deferring := false;
          (Now time out the Priority (contention) slots}
          for (i := nPriorities-1; i>=0; i--)
          begin
            slotTime := time();
            currentPriority := i;
            while (time()-slotTime < PRI_SLOT) do
               if    carrierSense()    then    endcycle;{restart
deference loop}
            {if priority slot passed with no contenders, then
that priority
             level must be idle, good practice says make sure
the backoff
             counters are reset}
            BL[currentPriority] := 0;
            MBL[currentPriority] := 0;
          end;
     end; {cycle}
end; {Deference}
{computeSignals: Determine which signals to send}
function computeSignals();
begin
     signalSlot   :=   -1;   {-1   means   no   signal   to   send,
initialization}
     if   (txReady   and   (txPriority   =   currentPriority)   and
BL[txPriority]=0) then
          signalSlot = integerRandom(nSignals); {select Backoff
Signal slot}
end; {computeSignals}

{processSignals:   Process the received signals, adjusting the
Backoff Levels}
```

```
function processSignals();
begin
     psignals := 0;
     for (i=0; i < nSignals; i++)
          if signal[i] then psignals++;
     if (txReady and (txPriority = currentPriority)) then
     begin
          backoffLevel := BL[currentPriority];
          if backoffLevel = 0 then
          begin
            tem := 0;
            for (i=0; i < signalSlot; i++)
               if signal[i] then tem++;
            BL[currentPriority] := saturate(0,nLevels-1,tem);
          end;
          if backoffLevel > 0 then
            if psignals > 0 then
               BL[currentPriority] :=
                 saturate(0,nLevels-1,backoffLevel   +   psignals-
1);
end;
if psignals > 0 then
     begin
          if  MBL[currentPriority]  =  0  then  MBL[currentPriority]
:= psignals;
          else    MBL[currentPriority]    =    saturate(0,nLevels-
1,MBL[currentPriority]
                                                + psignals-1);
     end;
end; {processSignals}
```

```
{Transmitter:  Wait for txReady and txPriority from the link
level process.
 send txFinished when frame has been sent.}
process Transmitter;
begin
     cycle
          while (not txReady) do nothing();
          BL[txPriority] := MBL[txPriority];
          while (not (txPriority >= currentPriority and
BL[txPriority]=0)
                 or deferring)
            do nothing();
          ttime := time();
          xmtDataOn(); {start data transmitting}
          while xmtBusy() and (time() - ttime < CD_FRAG) do
          begin
            if collisionSense() then
            begin
               xmtDataOff();{turn off, after sending minimum
collision fragment}
               Ncollisions++; {timeout on excessive collision
limit}
               if Ncollisions = attemptLimit-1 then
txFinished();
               endcycle;
             end;
          end;
          while xmtBusy() do nothing();
          txReady := false;
          txFinished();     {signal link level that frame has
been transmitted}
     end; { cycle }
```

```
end; { Transmitter }

{collisionSense: }
function collisionSense();
begin
     { When transmitting, detect the presence of a second
transmission.
       When receiving, detect overlapped transmissions}
end; { collisionSense }
{Receiver: }
process Receiver;
begin
     { Wait for carrier sense.  Demodulate received signals into
frames.
       Reject collision fragments. Determine frame boundaries.
Check FCS.
       Filter based on destination address. Perform optional
Link Layer
       signaling and other controller functions.}
end; { Receiver }
```